2004

Query Processing using Negative Tuples in Stream Query Engines

T. Ghanem

M. Hammad

M. Mokbel

Walid G. Aref
Purdue University, aref@cs.purdue.edu

Ahmed K. Elmagarmid
Purdue University, ake@cs.purdue.edu

Report Number:
04-030

QUERY PROCESSING USING NEGATIVE TUPLES IN STREAM QUERY ENGINES

T. Ghanem
M. Hammad
M. Mokel
W. G. Aref
A.K. Elmagarmid

CSD TR #04-030
November 2004
(REVISED APRIL 2005)

Query Processing Using Negative Tuples in Stream Query Engines

T. Ghanem[1] M. Hammad[2] M. Mokbel[1] W. G. Aref[1] A. Elmagarmid[1]
[1]Department of Computer Science, Purdue University.
[2]Department of Computer Science, University of Calgary.

Abstract

The concept of *Negative Tuples* (or delete tuples) has been adopted widely in data stream management systems to evaluate continuous sliding-window queries incrementally . The main idea is to produce a *negative* tuple for each expired tuple from the sliding window. Thus, various query operators can update their state based on the expired tuples. *Negative tuples* avoid the non-deterministic output delays that result from the commonly used *input-triggered* approach. However, *negative tuples* double the number of tuples through the query pipeline, thus reducing the pipeline bandwidth. In this paper, we put the *negative tuples* under a magnifying glass where we show its detailed realization in terms of its generation and its processing in various operators. Then, we present several optimization techniques that aim to reduce the overhead of the *negative tuples* approach. These optimizations can be applied independently or together to enhance the performance of *negative tuples*. A detailed experimental study, based on a prototype system implementation, shows the performance gains over the *input-triggered* approach of the *negative tuples* approach when accompanied with the proposed optimizations.

Keywords: Data stream management systems, pipelined query execution, negative tuples.

1 Introduction

The emergence of data streaming applications calls for new query processing techniques to cope with the high rate and the unbounded nature of data streams. The *sliding-window query model* is introduced to process continuous queries in-memory. The main idea is to limit the focus of continuous queries to only those data tuples that are inside the introduced *window*. As the window

slides, the query answer should be updated to reflect both the new tuples entering the window and the old tuples expiring from the window. Geared towards supporting continuous sliding-window queries, data stream management systems (e.g., [1, 2, 3, 12, 15, 24, 32]), provide the so-called *window-aware* operators. Window-aware operators (e.g., window-join and window-aggregates) are modifications of their counterpart traditional operators to support sliding-window queries. The main difference in window-aware query operators is the need to process tuples expired from the window as well as the newly incoming tuples.

Window-aware query operators have relied on the input-triggered approach (ITA for short) where the timestamp of the newly incoming tuples is used to expire old tuples [5, 28]. However, as will be discussed in Section 4.1, the ITA may result in significant delays in the query answer. As an alternative, the negative tuples approach (NTA for short) is introduced as a delay-based optimization framework that aims at reducing the output-delay incurred by the ITA [4, 26]. A *negative tuple* is an artificial tuple that is generated by an operator `EXPIRE` in the query pipeline (generalization for operator `SEQ-WINDOW` in [4] and operator `W-EXPIRE` in [26]). For each regular data tuple (i.e., *positive* tuple), say t, `EXPIRE` emits a corresponding *negative tuple* t^- once t becomes out of the sliding window. As the *negative tuple* flows through the query pipeline, it removes the effect of its corresponding *positive* tuple.

Although the basic idea of the NTA is attractive, it may not be practical. The fact that we introduce a negative tuple for every expired positive tuple means doubling the number of tuples through the query pipeline. Each *negative* tuple exactly repeats and undoes the effect of its corresponding *positive* tuple. In this case, the overhead of processing tuples through the various query operators is doubled. This observation opens the room for optimization methods to the basic NTA. Various optimizations would mainly focus on two issues: (1) Reducing the overhead of re-processing the *negative* tuples. (2) Reducing the number of the *negative* tuples through the query pipeline.

In this paper, we put the *negative tuples* under a magnifying glass where we show its detailed realization in terms of its generation and its processing in various operators. We show how in some cases the NTA improves the query performance. Then, we introduce the *join message* optimization that reduces the overhead of processing the *negative tuples* in the join operator. Furthermore, we introduce the *piggybacking* approach that *self-tunes* the query pipeline by alternating between the usage of ITA and NTA. Alternating between the two approaches is triggered by discovering

fluctuations in the input arrival rate that is likely to take place in streaming environments. In general, the contributions of this paper can be summarized as follows:

1. We compare the performance of the ITA and NTA for various queries. This comparison helps identify the appropriate situations to use each approach.

2. We give a detailed realization of the NTA in terms of its processing in various operators.

3. We introduce the *join message* optimization technique that aims to reduce the number of negative tuples through the query pipeline.

4. We introduce the *piggybacking* technique that aims to self-tune the query pipeline by alternating between the *input-triggered* and *negative tuples* approaches. This self-tuning feature allows the system to be stable with fluctuations in input arrival rates and filter selectivity.

5. We provide an experimental study using a prototype data stream management system that evaluates the performance of the *ITA*, *NTA*, *join messages*, and *piggybacking* techniques.

The rest of the paper is organized as follows: Section 2 highlights related work in data stream query processing. Section 3 gives the necessary background on the pipelined query execution model in data stream management systems. Section 4 discusses and compares the ITA and the ITA for pipelined execution of sliding window queries. Detailed realization of *negative tuples* in the various operators is given in Section 5. Optimizations over the basic NTA are introduced in Section 6. Section 7 studies the *piggybacking* technique. Experimental results are presented in Section 8. Finally, Section 9 concludes the paper.

2 Related Work

Stream query processing is currently being addressed in a number of research prototypes. Examples include Aurora [2], which is later extended to Borealis [1], NiagraCQ [15], TelegraphCQ [12], PSoup [14], NILE [24, 26] and STREAM [3]. These research prototypes address various issues in processing queries over data streams. All these research prototypes have recognized the need for sliding-windows to express queries over data streams. For a survey about the requirements for stream query processing, refer to [7, 18].

Window-aware query operators have been addressed many times in the literature. The previous work in this subject addresses the processing of a single window-aware operator but does not address the processing of a whole query pipeline. Unlike the NTA, window-aware query operators in [2] do not share the memory and processing among the overlapped portions of the sliding-windows. In [2], a computation state is initialized whenever a window is opened, that state is updated whenever a tuple arrives, and the state is converted into a final result when the window closes. In [25], the authors introduce classes of window joins where the window constraints are different among different pairs of input streams. In [28], the authors use the ITA to invalidate tuples from the join state when a new tuple arrives based on the window semantics. However, they do not address how to invalidate tuples for the operators above the join since an output from the join carries the semantics of two different windows. Multi-way window join is introduced in [19]. Memory-limited execution of window join is addressed in [34]. Window aggregates are another important class of operators [5, 2, 13, 17]. Window-independent semantics and evaluation techniques for aggregate queries have been proposed in [29].

The traditional query optimization goal is to reduce the query execution time. This goal does not apply to continuous queries. Optimization techniques over data streams have different goals. Rate-based optimization is introduced in [38]. The goal of the optimization is to maximize the output rate of a query. In [6], the authors introduce a framework for conjunctive query optimization. The goal of the optimization is to find an execution plan that reduces the resource usage. None of these optimization techniques use reducing the output delay as an optimization goal.

Since the characteristics of a data stream change continuously, adaptive query processing is necessary. Adaptive continuous queries are addressed in [30]. [39] introduces algorithms for dynamic plan migration for plans with statefull operators. Constraint-aware adaptive query processing is introduced in [32]. Proactive re-optimization is introduced in [9]. For a survey about adaptive query processing refer to [8].

Recent research efforts focus on introducing new "artificial" kinds of tuples that flow through the query pipeline. Examples of such tuples include *delete messages* [1], *DStream* [4], *Negative Tuples* [26], *heartbeats* [34], and *punctuation* [36]. The main idea of these artificial tuples is to notify various pipelined operators of a certain event (e.g., expiring a tuple, synchronizing operators, or end of sequence of data). STREAM [3] and Nile [24, 26] use NTA to expire tuples from window-aware

operators. Negative tuples have been used in other systems e.g., Borealis data stream management system [1] for automatic data revision. A negative tuple is sent by the streaming source to delete an erroneous positive tuple. Although, not mentioned explicitly, NiagraCQ [15] uses a notion similar to negative tuples when processing deletions to data streams. Punctuation is another form of artificial tuples [36]. A punctuation marks the end of a subset of the data. The difference between negative tuples and punctuation is that a negative tuple indicates the expiration of an old tuple. This old tuple should be re-processed and may change the query answer. On the other hand, a punctation purges tuples from the operators states because these tuples will not be further used in query evaluation. The purged tuples do not expire and do not need to be re-processed. Joining punctuated data streams has been addressed in [16]. Processing stream constraints is another way to discover and purge un-needed tuples from operators' states [10]. Stream constraints are generated using stream monitoring and stream summarizations. An operator-level heartbeat [34] is a way for time synchronization since a heartbeat is sent along the query pipeline so that the operators learn the current time and process input tuples accordingly. Heartbeats are processed only when there is an input in the operator's input queue and hence is input-triggered and does not address the output delay problem.

Processing negative tuples in the query pipeline to update the query answer is closely related to the traditional incremental maintenance of materialized views [22, 23, 11]. In the differential approach for incremental view maintenance, change propagation equations are defined for the various relational algebra operators [21]. The equation specifies how the operator should process an input (positive or negative) tuple.

3 Preliminaries

In this section we discuss the preliminaries for sliding-window query processing. First, we will discuss the semantics for sliding-window queries. Then, we will discuss the pipelined execution model for sliding-window queries over data streams.

3.1 Sliding-window query semantics

A sliding-window query is a continuous query over n input data streams, S_1 to S_n. Each input data stream S_j is assigned a window of size w_j. At any time instance T, the answer of the sliding-window query equals to the answer of the snapshot query whose inputs are the elements in the current window for each input stream. At time T, the current window for stream S_i contains the tuples arriving between times $T - w_i$ and T. The same notions of semantics for continuous sliding-window queries are used in other systems [31, 35]. In our discussion, we focus on time-based sliding-windows that are the most commonly used sliding-window type. The size of a time-based sliding-window query is determined in terms of time units.

Input data streams, S_1 to S_n, use a global clock to timestamp their tuples. Note that the timestamp is unique for each stream, i.e., no two tuples from the same stream can have the same timestamp. On the other hand, two tuples from different streams can have the same timestamp. The timestamp of the input tuple represents tuple arrival time (i.e., the time at which the tuple arrives to the system). The window w_i associated with stream S_i represents the lifetime of a tuple t from S_i. This means that, if a tuple t_i from stream S_i carries a timestamp ts, then tuple t_i should expire at time $ts + w_i$. Sliding-window query operators assume that the tuples are processed in an increasing order of timestamp ts.

Handling timestamps: A tuple t will carry two timestamps, t's arrival time, ts, and t's expiration time, Ets. Operators in the query pipeline handle the timestamps of the input and output tuples. The output of a unary operator (e.g., Select, Project, and Distinct) carries the same timestamp, ts, and expiration timestamp, Ets, as the input tuple. If an intermediate tuple in the pipeline is generated as a combination of more than one tuple (e.g., output of a join), the output tuple expires whenever any of its composing tuples expires. If a tuple t is generated from the combination of the two tuples $t1(ts1, Ets1)$ and $t2(ts2, Ets2)$, then t will have $ts = max(ts1, ts2)$ and $Ets = min(Ets1, Ets2)$. In the rest of the paper, we use the CQL [4] construct RANGE to express the size of the window in time units.

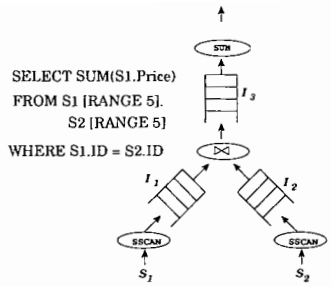


Figure 1: Stream queuing model.

3.2 Data stream queuing model

Data stream management systems use a pipelined queuing model for the coordination among different query operators [4]. The queuing model of data streams is a modification of the one used in traditional database management systems [20]. All query operators are connected via first-in-first-out queues. Each operator, say p, has its own input queue I_p that receives the incoming input tuples to the operator p. p is scheduled once there is at least one data input in its input queue I_p. Upon scheduling, p processes its input and produces an output result. The output of p is either sent as a query result to the user (if p is the top operator in the pipeline) or is inserted in p's output queue, which is the input queue of the next operator in the pipeline.

The bottom stream operator in any continuous query pipeline over data streams is the SCAN operator. The stream SCAN (SSCAN) operator acts as an interface between the streaming source and the query pipeline. SSCAN is responsible for reading tuples from the streaming source. Each stream source S_i will have an SSCAN operator. The newly incoming tuples are inserted into the input queue of the first operator in the pipeline. SSCAN assigns to each input tuple two time stamps, ts which equals the tuple arrival time, and Ets which equals $ts + w_i$. No two tuples from the same stream can have the same timestamp ts. Incoming tuples are processed in increasing order of their timestamps. Figure 1 gives a sliding-window query and the corresponding pipelined query plan. The window size for each input stream is five time units. Notice that queue I_3 is both the input queue for the SUM operator and the output queue for the join operator. Some query operators (e.g., join, aggregates, and distinct) are required to keep some state information. Basically, in continuous sliding window queries, such operators need to keep track of all tuples that have not expired yet. However, the maintenance of up-to-date state information is challenging.

Traditional pipelined query operators are designed mainly to deal with data resident on disk. To

cope with the streaming queuing model, traditional pipelined query operators should be modified (e.g., see [28]). In addition to processing the new incoming tuples, windowed-pipelined query operators need to address three main challenges:

1. Updating the state information of each query operator. State information needs to be updated by expiring (i.e., deleting) old tuples that become outside of the sliding-window of the corresponding stream as well as adding the new incoming tuples. The challenge is how to trigger the change of status. An operator may continuously check the time in the lower operators in the pipeline to decide about expiring some data. This continuous checking for time is a CPU intensive task.

2. The action to be taken when a tuple expires. Once an operator decides to expire (delete) one of the input tuples from the operator's state, the query operator needs to know what to do with the expired tuples. The required action could be just dropping the expired tuple, or reprocessing the expired tuple and producing a new answer. Processing expired tuples by the various query operators will be discussed in detail in Section 5.

3. Forwarding the effect of the expired tuples to the next operator in the query pipeline.

Query operators handle these three issues differently depending on the operator's semantics. For correct and efficient execution of the query, window-aware query operators should be designed to address these three challenges efficiently.

4 Pipelined-execution of Sliding-window Queries

In this section, we discuss the two approaches for pipelined execution of sliding-window queries, namely the *input-triggered* and the *negative tuples* approaches. We discuss how each approach addresses the three challenges presented in Section 3.2 along with the drawbacks of each approach.

4.1 The input-triggered approach (ITA)

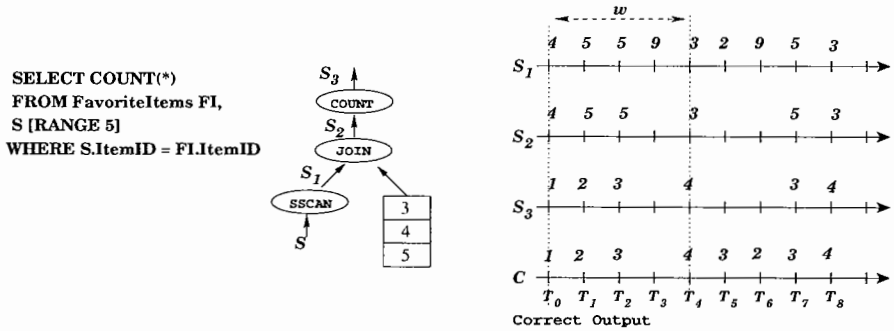
The main idea of the commonly used *input-triggered* approach is to schedule different query operators only when they have input tuples in their input queues. Thus, an incoming input tuple triggers

the processing of a query operator. Once an operator starts to process an input tuple, the operator knows the current time T from the tuple's timestamp. Processing a new input tuple includes two main tasks: (1) expiring tuples based on the current time, the query operator can decide which of the tuples in its state information (if any) are expired. This can be performed by scanning all tuples in the operator's state and removing any tuples with expiration timestamp, Ets, less than T. (2) processing the new tuple itself.

The action to be taken for the expired tuples is operator-dependent. For example, aggregate operators (e.g., SUM, COUNT, and MAX) need to process the expired tuples as well as the new incoming tuples. For example, in the COUNT operator, the number of expired tuples should be subtracted from the answer and the new answer should be reported. Processing expired tuples by the various query operators will be discussed in detail in Section 5. The output of the operator will carry the current time information so that the next operator in the pipeline can behave accordingly.

The ITA has a major flaw where it can result in a non-deterministic delay in the query answer. In a streaming environment, a delay in updating the answer of a continuous query is not desirable and may be interpreted by the user as an erroneous result. The delay in the query answer comes form the fact that unlike traditional continuous queries where the query answer changes only with the arrival of new input data, the answer of *continuous time-based sliding window queries* depends on the query time as well as on the input to the query. By being *time-dependent*, the answer of a *sliding-window* query may change even if no new input arrives to the outstanding query. This is because some tuples may expire.

For example, consider the query Q_1 *"Continuously report the number of favorite items sold in the last five time units"*. The identifiers for favorite items are stored in the table *FavoriteItems*. This query includes a join between the input stream and the relational table *FavoriteItems*. Notice that even if the input is continuously arriving, the join operator will filter out many of the incoming stream tuples and hence the upper operators in the pipeline will not get any notification about the arrival of a new tuple. This problem arises because of the existence of the highly selective operator (join) at the bottom of the query pipeline. The query answer will not be updated until one tuple passes the join. The same problem may happen even if there are no sales for a certain time period T, the query answer may change because some parts of the answer become too old (i.e., more than five time units old).



(a) Query Q1 with the query piepline

(b) Execution time line

Figure 2: Input-triggered evaluation.

Figure 2 illustrates the behavior of the ITA for Q_1. The time lines S_1 and S_2 correspond to the input stream and the output of the join operator, respectively. S_3 and C represent the output stream using the ITA and the correct output, respectively. The window w is equal to five time units. Up to time T_4, Q_1 matches the correct output C with the result 4. At T_5, the input "2" in S_1 does not join with any item in the table *FavoriteItems*. Thus, the COUNT operator will not be scheduled to update its result. Thus, S_3 will remain 4 although the correct output C should be 3 due to the expiration of the tuple with value 4 (which arrived at time T_0). Similarly, at T_6, S_3 is still 4 while C is 2 (the tuple with value "5", which arrived at time T_1, has expired). S_3 keeps having an *erroneous* output till tuple "5" is received at time T_7 where it triggers the scheduling of the COUNT operator to produce the correct output "3". This erroneous behavior motivates the idea of having a new technique that triggers the query operators either based on input change or time change.

4.2 The negative tuples approach (NTA)

The main goal of the *negative tuples* approach is to separate tuple expiration from the arrival of new input tuples. The main idea is to introduce a new type of tuples, namely negative tuples, to represent expired tuples [4, 26]. The approach distinguishes between two types of tuples: *Positive* and *Negative* tuples. *Positive* tuples are those regular tuples that are received from the input streams. *Negative* tuples are a special kind of tuples that are produced from the query pipeline itself. A special operator `EXPIRE` will be added at the bottom of the query pipeline. `EXPIRE` will emit a negative tuple for every expired *positive* tuple. A negative tuple is responsible for cancelling

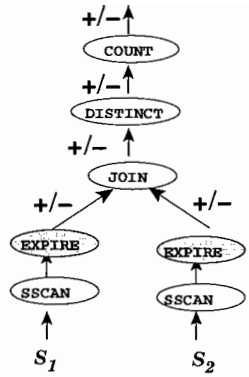


Figure 3: The negative tuples Approach.

the effect of a previously processed positive tuple. For example, in time-based sliding-window queries, a positive tuple t^+ with timestamp T from stream I_j with window of length w_j, will be followed by a negative tuple t^- at time $T + w_j$. The negative tuple t^- will be processed by the various operators in the query pipeline. Upon receiving a negative tuple, each operator in the pipeline will behave accordingly to expire the effect of the old tuple. Various query operators will process negative tuples in the following ways:

- *Selection* and *Join* operators handle *negative* tuples in the same way as *positive* tuples. The only difference is that the output will be in the form of a *negative* tuple. Optimizations are introduced in Section 6.1 to avoid re-performing the join for negative tuples.
- *Aggregates* update their aggregate value by dropping the positive tuple corresponding to the received *negative* tuple and output the new aggregate value.
- The *Distinct* operator reports a *negative* tuple at the output only if the corresponding *positive* tuple is in the results reported recently .

In Section 5, we give details about processing negative tuples in various query operators. The NTA is designed with the following goals in mind:

1. Providing a scheme that efficiently coordinates among various pipelined query operators.
2. Avoiding the delay in updating the continuous sliding-window query answer.
3. Providing a scheme that is practical and general enough to be easily implemented.

The first goal is achieved by introducing the notion of *negative tuples*. The second goal is achieved by the fact that the newly introduced *negative* tuples synchronize the pipeline so that there is no need to wait for any input to trigger the processing. The third goal is achieved by encapsulating the window semantics in a new operator EXPIRE. Then, traditional pipelined operators are modified to accommodate the concept of *negative tuples* independent from the window semantics. Figure 3 gives an example of a query plan using the NTA. Note that all operators in the pipeline read negative tuples, process them, and may emit negative tuples as output.

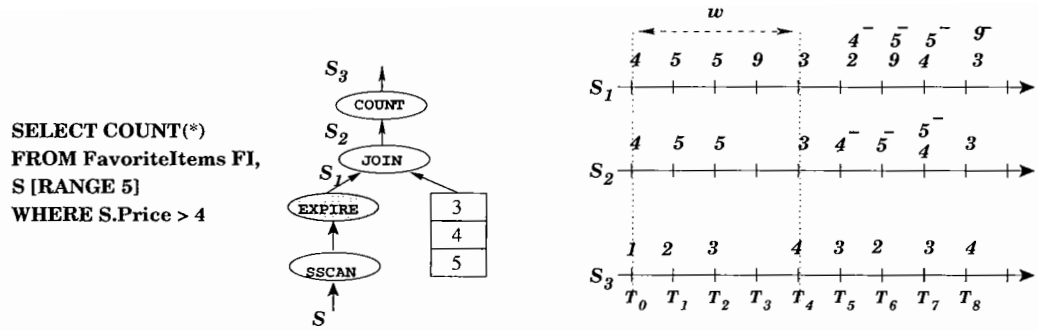
4.2.1 Negative tuples generation:

A special operator, termed EXPIRE, is attached with every input stream (can be merged with the SSCAN operator). Every tuple t in the query plan will have a flag associated with it to indicate whether it is a positive or a negative tuple. Positive tuples are the input stream tuples. Whenever a new tuple is read from the stream, it is assigned a positive sign then it is sent to the query pipeline to be processed. Two timestamps, timestamp ts, and expiration timestamp Ets, are associated with the positive tuple. As explained in Section 3.1, when a tuple t from stream I_j arrives at time T, it will have $ts = T$ and $Ets = T + w_j$, where w_j is the size of the window assigned to stream I_j. EXPIRE will buffer the input stream tuples in the current window. When a tuple should expire, EXPIRE will emit a negative tuple that is exactly the same as the positive tuple but with a negative sign assigned to it. The negative tuple timestamp, ts, is equal to the expiration timestamp, Ets, of the corresponding positive tuple. the negative tuple does not have an expiration timestamp. Negative tuples are processed by the query pipeline operators, as explained in Section 5.

4.2.2 Handling delays using negative tuples

Figure 4b gives the execution of the NTA for the example in Figure 4a (the negative tuples implementation of the query in Figure 2a). At time T_5, the tuple with value 4 expires and appears in S_1 as a negative tuple with value 4. The tuple 4^- joins with the tuple with value 4 in the *FavortieItems* table as it follows the footsteps of tuple 4^+. At time T_5, the COUNT operator receives the negative tuple 4^-. Thus, COUNT outputs a new count with value 3. Similarly at time T_6, COUNT receives the negative tuple 5^-. Thus the result is updated to 2.

The previous example shows that the NTA overcomes the output delay problem introduced by



(a) Query Q1 with the query pipeline (b) Execution timeline

Figure 4: Negative tuples evaluation.

the ITA. In the NTA, tuple expirations are independent from the query characteristics. This means that, even if the query has highly selective operators at the bottom of the pipeline, we can produce timely correct answers. On the other hand, if the bottom operator in the query pipeline has low selectivity, then it will pass almost all input tuples to the intermediate queues in the pipeline. In this case, the NTA may present more delays due to increasing the waiting times in operator queues.

4.3 Invalid tuples

In the ITA, negative tuples are not explicitly generated for every expired tuples but some invalid (negative) tuples may be generated internally by operators in the query pipeline. The set-difference operator is an example operator that may generate invalid tuples. Assume the input to the set-difference operator is two streams S_1 and S_2 and the output should be the stream $S - R$. At time T_1, a tuple t_1 with value 9 arrives in stream S_1. t_1 is output in the set-difference between two streams S_1 and S_2 (9 appears in S_1 but not in S_2). Later, at time T_2 ($T_2 < T_1 + w_1$), tuple t_2 with value 9 arrives in stream S_2. Tuple t_1 expires from the output of the set-difference operator although t_1 does not expire from the sliding-window w_1. This example shows that even in the ITA, various query operators should be furnished by methods to process negative tuples.

5 Window Query Operators

Window query operators differ from traditional operators in that window query operators need to process expired tuples as well as newly incoming tuples. There are three sources for expired

tuples: (1) An explicit negative tuple (e.g., the ITA). (2) A tuple in the operator's state that has an expiration timestamp, Ets, that is less than the input tuple timestamp (e.g., the ITA). (3) An invalid tuple that is generated by an internal operator in the query pipeline (e.g., the set-difference operator). The actions to be taken by the operator to process the expired tuples is the same for the three cases and is operator-dependent. In this section, we discuss how the various query operators process the expired tuples. We will use the terms expired tuple and negative tuple interchangeably.

5.1 Classification of window operators

Relational query operators can be classified into four classes based on the type of the processed tuple and the type of output tuple as follows:

- *Case* 1: A *positive* tuple, t^+_{out}, is produced as a result of processing a *positive tuple*, t^+_{in}.
- *Case* 2: A *negative* tuple, t^-_{out}, is produced as a result of processing a *positive* tuple, t^+_{in}.
- *Case* 3: A *positive* tuple, t^+_{out}, is produced as a result of processing a *negative* tuple t^-_{in}.
- *Case* 4: A *negative* tuple, t^-_{out}, is produced as a result of processing a *negative* tuple, t^-_{in}.

Cases 1 and 4 can arise in all window operators. For example, consider the $\widehat{\text{SELECT}}$ operator. For Case 1, when $\widehat{\text{SELECT}}$ receives an input tuple t^+_{in}, if t^+_{in} satisfies the selection predicate then a new tuple t^+_{out} will be produced. For Case 4, when a negative tuple t^-_{in} becomes an input to the $\widehat{\text{SELECT}}$ operator, and assuming that the corresponding positive tuple t^+ results in an earlier output, then the tuple t^-_{out} should be produced. t^-_{out} indicates that the corresponding positive tuple is no longer part of the current window. Cases 2 and 3 are special to some window operators as will be explained in the following sections.

In the following sections, we assume that the operators process the input tuples in increasing order of the timestamp, ts. Tuples arriving to the system in an out-of-order fashion can be stored in buffers and ordered using heartbeats [33]. Ordering tuples is out of the scope of this paper.

5.2 Window Distinct

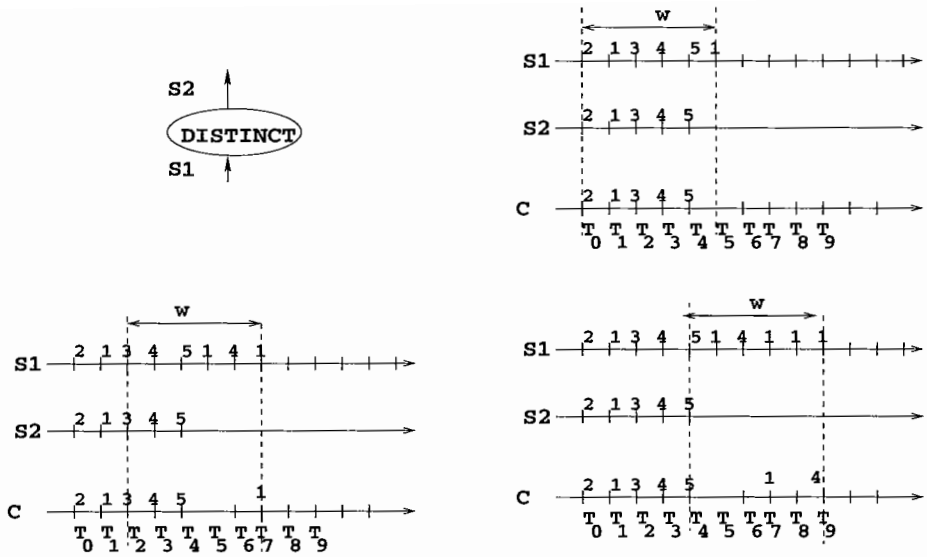


Figure 5: Unexpected answers from the Distinct operator

Conventional approaches for duplicate elimination do not consider tuple expiration and hence can produce *incorrect* output in sliding-window queries as shown in the following example.

Example The example in Figure 5 gives a sliding window query that contains a windowed Distinct operation. The input stream $S1$ has a window of size six time units. $S1$, $S2$, and C represent the input stream, the output stream after the Distinct operator, and the correct output from the Distinct operator, respectively. $S2$ reports correct answers until time T_5. However, at time T_7, tuple 1 expires from $S1$ and hence it expires from $S2$. Since tuple 1 is one of the distinct tuples of Stream $S2$, the output of Stream $S2$ at time T_2 does not reflect the correct distinct values (compared to Stream C). Similarly, at time T_9, tuple 4 expires from Stream $S1$ and $S2$ and the distinct tuple in $S2$ (a single tuple 5) does not reflect the correct distinct tuples at T_9 (the distinct values at time T_9 are the tuples 5, 1, and 4). This incorrect output of the Distinct operator results from ignoring the effect of tuple expiration. As the window slides, the input tuples in $S1$ and their corresponding output tuples in $S2$ are expired. While the expired output tuples are still duplicates in the current window, $S2$ provides no new output tuples to replace the expired output tuples. Thus, the output of Distinct becomes erroneous.

The W-Distinct Algorithm Similar to the traditional hash-based duplicate-elimination algorithm [20], the windowed Distinct algorithm (W-Distinct) compares an input positive tuple t_n^+ with the set of previously received tuples (stored state). Assume that the Distinct operation is on the

attribute *DSTAttr* of the input tuples. If an input tuple, say t_n^+, has a distinct value of *DSTAttr*, W-Distinct inserts t_n^+ in the output stream and adds t_n^+ to the stored state. W-Distinct outputs a new tuple t to replace an expired tuple t_e, whenever t_e is produced before as a distinct tuple and t is a duplicate (i.e., has the same value of the attribute *DSTAttr*) for t_e. Typically, this is Case 3 that is presented in Section 5.1.

The Algorithm W-Distinct uses the following two data structures: (1) A hash table, H, to store the distinct tuples in the current window and. Tuples are hashed into H based on the values of the *DSTAttr* attribute. (2) a sorted list, Distinct List (or DL for short), to store all output distinct tuples sorted by their timestamp. Given these data structures, Algorithm 1 illustrates the steps of the W-Distinct operator. Expired tuples in Step 1 of the algorithm are identified either when the expiration timestamp of the tuple is less than the timestamp of the input tuple (input-triggered) or when a negative or invalid tuple is received.

Example Consider the example in Figure 5 when using the W-Distinct Algorithm. At time T_7, tuple 1 of Stream $S2$ expires and the condition in Step 3 of Algorithm 1 is True. Therefore, Steps 4-7 of W-Distinct will produce a new output tuple, 1, which represents the correct answer as in Stream C. Similarly, W-Distinct produces a correct output at time T_9 when tuple 4 expires.

5.3 Window set operations

The Window Union (W-Union) operator is straightforward and can be implemented using the traditional Union operator with little modifications. W-Union must process input tuples from different sources in-order (increasing timestamp) and expire the old tuples. On the other hand, the window set-difference (W-Minus) and the window Intersect (W-Intersect) operators are quite involved. W-Intersect has similar cases as those of W-Group-By (see Section 5.4). Therefore, we choose in this section to present the W-Minus operator only.

The W-Minus algorithm The W-Minus between streams S and R produces in the output stream tuples in S that are not included in R during the last windows of streams S and R. W-Minus can produce a negative output tuple as it receives a positive input tuple (*Case* 2). For example, consider a tuple t_s from Stream S that has no duplicates in R and t_s was reported in the set difference set. When a tuple t_r arrives in stream R and t_r is a duplicate for t_s, then t_s must be reported as a negative tuple in the output. Furthermore, W-Minus can produce new positive

Algorithm 1 The W-Distinct Algorithm
Input: *t: An incoming tuple. H, DL: Two hash tables that represnet the state of the input and output distinct tuples, respectively.*
Algorithm

```
1)  If t is an expired tuple
2)      Remove t from H
3)      If t is found in DL /*t was distinct*/
4)          Remove t from DL
5)          Add t⁻ to the output stream
6)          Probe H using the values
            of DSTAttr in t
7)          If a tuple t* is found such that t* has
            the same values of DSTAttr as t
            /*A duplicate of the expired
            tuple exists in the window*/
8)              Add t* to DL and the output stream.
9)  Else If t is a positive tuple
10)     Probe H using the values of DSTAttr in t
11)     If no matching tuple is found in H
        /* Tuple t is distinct */
12)         Add t to DL and to the output stream
13)     Add t to H
```

Algorithm 2 The W-Minus Algorithm
Input: *t: An incoming tuple. H_S, H_R, F_S, F_R and O_S: Tables represent the W-Minus operator state.*
Algorithm

```
1)  If t is an expired tuple, from stream S or R
2)      If t ∈ stream S
3)          Remove t from H_S and update f_s(t) in F_S
4)          Retrieve f_r(t) from F_R
5)          If f_s(t) >= f_r(t)
6)              If t ∉ O_S
7)                  Remove from O_S tuple t_i with the same Attr as t
8)                  Add t_i⁻ to the output stream
9)              Else Remove tuple t from O_S and output t⁻
10)     Else If t ∈ stream R
11)         Remove t from H_R and update f_r(t) in F_R
12)         Retrieve f_s(t) from F_S
13)         If f_s(t) > f_r(t)
14)             Retrieve from H_S tuple t_i with the same Attr as t
15)             Add t_i to the O_S and to the output
16) If t is a positive tuple at stream S
17)     Add t to H_S and update f_s(t) in F_S
18)     Retrieve f_r(t) from F_R
19)     If f_s(t) > f_r(t)
20)         Add t to the O_S and to the output
21) If t is a positive tuple at stream of R
22)     Add t to H_R and update f_r(t) in F_R
23)     Retrieve f_s(t) from F_S
24)     If f_s(t) ≥ f_r(t)
25)         Remove from O_S tuple t_i with the same Attr as t
26)         Add t_i⁻ to the output stream
```

output tuple when a previously input tuple expires (*Case* 3). Consider an expired tuple t_r from Stream R that has no duplicates in R. In addition, t_r has a duplicate tuple in Stream S. When tuple t_r expires, the duplicate tuple of t_r in Stream S must be reported as the new output tuple.

The W-Minus algorithm given here is duplicate-preserving (i.e., Minus ALL). The duplicate-free version of the operator can be easily implemented by following the W-Minus with a W-Distinct operator. We adopt the SQL definition of duplicate-preserving Minus operator, where duplicates are significant in each stream (e.g., if stream S has n duplicates of tuple a and R stream has m duplicates of tuple a, the output is $max(0, n - m)$ duplicates of tuples a).

The Algorithm uses the following auxiliary structures: (1) Hash tables (H_S and H_R) to store the input tuples from streams S and R, respectively, (2) Frequency tables (F_S and F_R) to store the number of occurrences of each distinct tuple in S and R, respectively. We use $f_s(t)$ and $f_r(t)$

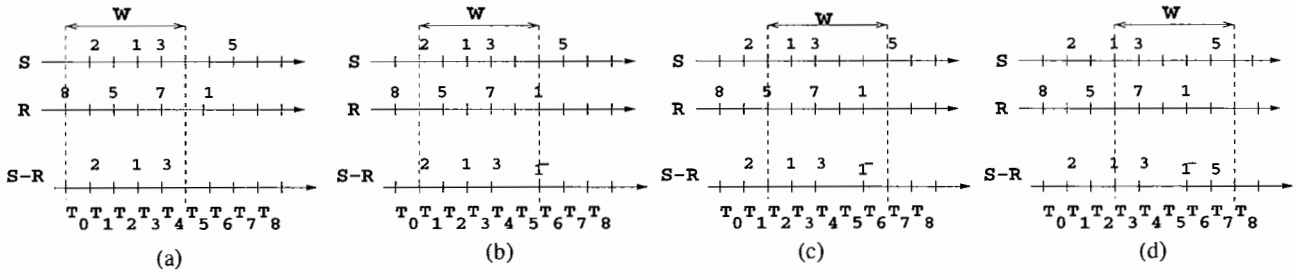


Figure 6: W-Minus Example.

to represent the count of duplicates for tuple t in Streams S and R, respectively. (3) Output table (O_S) to store the output tuples from Stream S that have not expired yet. Given these auxiliary structures and an input tuple with attributes $Attr$, Algorithm 2 presents the details of the W-Minus operator for sliding window queries. Notice that although W-Minus is a binary operator, the output tuple carries the information from only one stream (unlike the join).

Example. The example in Figure 6 shows the behavior of the W-Minus algorithm. The window size for each stream is six time units. Up to time T_5, Stream S contains the tuples $2, 1$, and 3 ($f_s(2) = f_s(1) = f_s(3) = 1$). Since no corresponding tuples for $2, 1$, and 3 exist in Stream R, the tuples are reported in the output stream $S - R$ (Steps 16-20 of the W-Minus Algorithm) (Figure 6a). At time T_6, tuple 1 arrives in Stream R ($f_r(1) = 1$). Since 1 has appeared in Stream S and has been reported in the output Stream (as part of the difference set), the W-Minus produces 1^- tuple to invalidate tuple 1 in the output stream (Steps 21-26)(Figure 6b). At time T_7, tuple 5 arrives in Stream S. Since tuple 5 exists in Stream R, the tuple is not reported in Stream $S - R$ (Figure 6c). However, at time T_8, tuple 5 of Stream R expires ($f_r(5) = 0$). At this time, tuple 5 of Stream S should belong to the difference set. Therefore, at time T_8, tuple 5 is reported at the output stream (Steps 10-15 of the Algorithm) (Figure 6d).

5.4 Window Aggregates and Group-By

Similar to W-Distinct, the correct execution for window aggregate (W-Aggregate) and window Group-By (W-Group-By) may produce a new positive output when a tuple expires (Case 3). This is the case since the aggregate operation represents a function over a set of tuples (e.g., SUM), changing this set (either by expiration or by addition) usually invalidates the value of the previous output and produces a new output. We focus in this Section on the W-Group-By operation as

W-Aggregate is a special case of W-Group with a single group.

We present the W-Group-By algorithm while considering a general execution framework that can support any aggregate function (e.g., SUM, COUNT, MEDIAN ...etc). Attributes of the input tuples can be divided into two classes G and A. G represents the values of the grouping attributes and A represents the values of the aggregate attributes (attribute a_i belongs to A if a_i appears in the $AggrFn_1(a_1)...AggrFn_n(a_n)$ list of the SQL Select clause). ts is the timestamp of the input tuple. For simplicity, we use F to represent the aggregate functions $AggrFn_1(.),...,$ $AggrFn_n(.)$. We use V to refer to the set of results after applying function F on the group tuples. The W-Group-By algorithm uses the following auxiliary structures: (1) GroupHandle, GH_i (one for each group): stores the state of the current group such as current aggregate values and the timestamp of the group that is the maximum timestamp of all tuples in the group. (2) Hash table, H: stores all tuples in the current window hashed by values in their grouping attributes. An entry in H stores the tuple and the corresponding GH_i.

The W-Group-By Algorithm Algorithm 3 presents the details of the W-Group-By operator for sliding window queries. Steps 1 to 9 handle the expired tuples, Steps 5 and 6 produce a new output if the expired tuple belongs to a group that still contains non-expired tuples. Steps 8 and 9 produce a NULL value for the empty group. Input positive tuples are handled by Steps 10 to 16 of the Algorithm. Step 12 creates new group for new tuples that do not belong to any of the current groups. Steps 15 to 16 compute a new aggregate value and produce a new group value. The timestamps of the output group is the timestamp of the input tuple. Notice that the W-GROUP-By operator will output a new tuple for each group whenever a tuple enters into or expires from the group. The output of the group will carry the timestamp of the most recent tuple added to the group. A group will expire when all its tuples expire.

5.5 Window Join (W-Join)

For joining data streams, a symmetric evaluation is more appropriate than the fixed-outer evaluation since both sides of the join may act as outer to perform the join. The extension of the symmetric approach for W-Joins over data streams is presented in [19, 25, 28].

The W-Join Algorithm W-Join needs to address only *Cases* 1 and 4. W-Join never produces a new positive output as an input tuple expires nor does it produce an invalid output. The join

Algorithm 3 The W-Group-By Algorithm
Input: *t: An incoming tuple. GH, H: hash tables represent the GROUP-BY operator state.*
Algorithm

```
1)  If t is an expired tuple
2)      Remove t from H
3)      Probe H with the grouping attribute G in t
4)      If found a matching entry
        /* The group has non-expired tuples and
        should report new aggregate values */
5)          Apply F on tuples in group G to get V.
6)          Add < G, V, t.ts > to the output stream
7)      Else /* The Group expires */
8)          Delete GH
9)          Add < G, NULL, t.ts > to the output
10) If t is a new tuple at the input stream
11)     Probe H with values of G in t
12)     If not found a matching entry
        /* Tuple < G, A, t.ts > forms a new group */
13)         Create GH for G
14)         Add (< G, A, t.ts >, GH) to H
15)     Apply F in the tuples in group G to get V.
16)     Add < G, V, t.ts > to the output stream
```

Algorithm 4 The W-Join Algorithm
Input: *t_i : An incoming tuple from stream S_i. H_1, H_2: Two hash tables for S_1 and S_2 that represent the join operator state.*
Algorithm

```
1)  If t_i is a positive tuple
2)      B_x = hash(t_i)
3)      Insert t_i in bucket B_x in the hash table H_i
4)      For each tuple t_j in
        bucket B_x in the other hash table
5)          If t_j joins with t_i
6)              output a positive join output tuple
                t+ for (t_i and t_j) with:
7)              t+.ts = max(t_i.ts.t_j.ts)
8)              t+.Ets = min(t_i.Ets.t_j.Ets)
9)  Else if t_i is an expired tuple
10)     B_x = hash(t_i)
11)     Delete tuple t_i from bucket B_x
12)     For each tuple t_j in
        bucket B_x in the other hash table
13)         If t_j joins with t_i
14)             output a negative join output tuple
                for (t_i and t_j) with timestamp = t_i.ts
```

processes expired tuples in the same way as it processes the input positive tuple. The input tuple is joined with the qualifying tuples from the other stream and the corresponding positive or negative output tuples are produced. Notice that the output of the join operator carries the semantics (windows) of two different streams [39]. W-Join needs to process tuples in increasing timestamps and assigns timestamps for its output tuples as follows: the timestamp, ts, equals the maximum value of the timestamps for all joined tuples. The expiration timestamp, Ets, equals the minimum value of expiration timestamps for all joined tuples (output of the join should expire whenever any of its composing tuples expire). Algorithm 4 gives the pseudo code for the symmetric hash join when considering expired tuples.

5.6 Window Project (with duplicates) and window Select operators.

The Select and project operators are non-statefull operators. A *negative* tuple t^- is processed in the same way as a *positive* tuple t^+. The difference is that when processing a negative tuple the output will be a negative tuple as well.

5.7 Result interpretation

The output of a sliding-window query is a stream of positive and negative tuples. Each negative tuples cancels a previously produced positive tuple with the same attributes. The output stream can be directed immediately to the query sink and the user will be responsible for constructing the net result of the query.

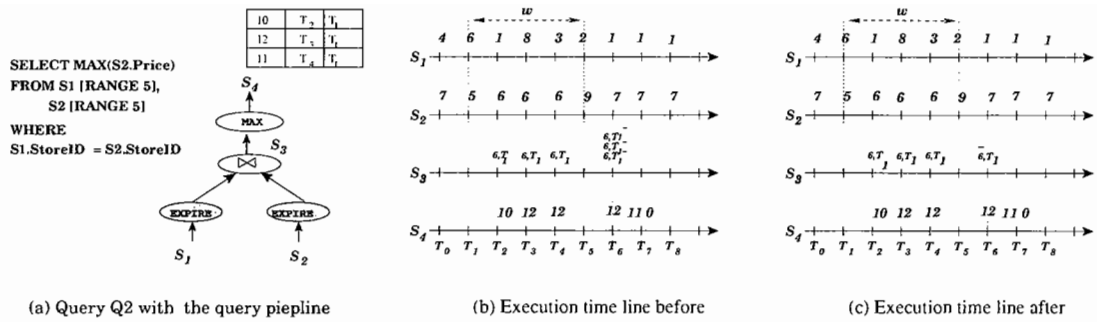
Another alternative for query output is to include a root operator, called the *recipient*, that collects the result from the sliding window query in a temporary buffer (state). When a positive tuple, say t^+, arrives, the recipient operator appends t^+ to the state. When a negative tuple, say t^-, arrives the recipient operator deletes the corresponding tuple t^+ from its state. The *recipient* operator can emit the net result of the query periodically or whenever being pulled by the user.

6 Negative Tuples Optimizations

Although the basic idea of the NTA (as described in Section 4.2) is attractive, it may not be practical. The fact that we introduce a negative tuple for every expired tuple through the query pipeline may result in doubling the number of tuples through the query pipeline. For example, for each incoming *positive* (regular) tuple, the `EXPIRE` operator produces its corresponding *negative* tuple. Doubling the number of tuples through the query pipeline is cumbersome if each *negative* tuple exactly repeats and undoes the effect of its corresponding *positive* tuple. In this case, the overhead of processing tuples through the various query operators is doubled. This problem is worst in the case of join subtrees in a query evaluation plan.

This observation gives rise to optimization methods of the basic NTA. Various optimizations would focus mainly on two issues: (1) Reducing the overhead of re-processing the *negative* tuples. A query operator should be "smart" enough to remember or at least keep some hints from the processing of each *positive* tuple t^+ to avoid a complete re-processing of the corresponding *negative* tuple t^-. (2) Reducing the number of *negative* tuples through the query pipeline. These optimizations can be addressed either at each operator level, or at the source operator `EXPIRE`.

For the rest of this section, we propose the *join message* technique to reduce the overhead of processing negative tuples in the join operator. The proposed technique addresses the two issues discussed above. The *join message* aims to avoid re-joining a *negative* tuple and hence reduces the



(a) Query Q2 with the query piepline (b) Execution time line before (c) Execution time line after

Figure 7: The Join Message Technique.

overhead of handling the negative tuples as well as reduces the number of *negative* tuples emitted from the *join* operator. As will be explained within the experimental results in Section 8, the *join message* optimization technique enhances the performance of the NTA.

6.1 The Join message optimization

In this section, we present the join message optimization for reducing the overhead of negative tuples in the query pipeline. The join message optimization is specific to the join operator, which is one of the most expensive operators in the query pipeline. Without this optimization, the join operator would normally re-process the negative tuples in the same way as their corresponding positive tuples. Given the highly expensive cost of the join operation, the join message technique aims to avoid re-processing of the negative tuples. Instead, the join operator will keep some state information in which the newly coming negative tuple can use to avoid the join processing. Then, once a negative tuple arrives to the join operator, a join message will be *passed* directly to the above query operator without performing the join and hence avoids the high cost of join processing.

6.1.1 Algorithm and Data Structures

Upon receiving a positive tuple, the join message optimization acts exactly as the traditional join algorithm in Algorithm 4. However, upon receiving a negative tuple, instead of re-performing the join operation, the join message optimization performs the following steps: (1) Remove the corresponding positive tuple from the join's state, (2) Set a special flag in this tuple indicating that this tuple is a join message (3) Put the join message in the join's output queue.

When the operator above the join receives a negative tuple with the join message flag set, the

operator learns that all positive tuples that resulted previously from joining with this tuple are expired and acts accordingly. This can be achieved by scanning the operator's state and expiring all tuples that carry the same expiration timestamp (Ets) as the join message. If the operator's state is sorted on the Ets attribute of the tuples, then this scan should not be costly. If the operator above the join does not carry a state (e.g., a Select), then that operator will just pass the join message to its output queue. If the join operator is the last operator in the pipeline, then the join message is sent directly to the output stream. The join message in this case is interpreted as a negative tuple that expires a group of previously emitted positive tuples (the positive tuples having the same expiration timestamp as the join message).

The join message optimization is designed with two goals in mind: (1) Reduce the work performed by the join operator when processing a negative tuple and (2) Reduce the number of negative tuples emitted by the join operator. Note that the join message achieves its goals as follows: (1) The negative tuples are "*passed*" through the join operator without probing the other hash table. (2) Only one negative tuple is emitted for every processed negative tuple independent from its join multiplicity. A large number of negative tuples can be avoided in the case of one-to-many and many-to-many join operations. One-to-many and many-to-many join operations are common in stream applications, for example, in on-line auction monitoring [34].

6.1.2 Example

Figure 7 gives an example of the join message approach. Figure 7a is the query pipeline. Two input streams $S1$ and $S2$ are joined. Both streams have the same input schema: <ItemId, Price, StoreID>. The sliding-windows for the two streams are of the same size and are equal to five time units. Notice that in the sliding-window model, aggregate operators maintain a state containing the tuples in the current window. In the figure, the table beside the MAX operator gives MAX's state. The table consists of three columns: the first column is for the value used in the MAX aggregation (S2.Price), and the second column is for the tuple timestamp and the third column is for the tuple expiration timestamp (other attributes may be stored in the state but are omitted for clarity of the discussion). Figure 7b gives the stream of tuples in the pipeline when using the NTA and before applying the join message optimization. The values on the lines represent the joining attribute (StoreID). Figure 7c gives the stream of tuples in the query pipeline after applying the

join message optimization. A tuple with joining attribute value 6^+ arrives at Stream S_1 at time T_1. Three subsequent tuples from S_2 (at times T_2, T_3 and T_4) join with the tuple 6^+ (at time T_1) from Stream S_1. The output of the join will have an expiration timestamp of the tuple that will expire first from the two joining tuples. In this example, the output of the join will carry expiration timestamp T_1 since tuple 6^+ from S1 will expire first. At time T_6, tuple 6^+ from Stream S_1 expires. In the NTA (Figure 7b), the join operator will perform the join with tuple 6^- and output three output negative tuples. The three tuples are processed by the MAX operator independently. As mentioned in Section 5.4, the MAX operator will output a new output after processing each input tuple (positive or negative). When applying the join message optimization, (Figure 7c), the join operator sends a join message with timestamp T_1 to its output queue. Upon receiving the join message, the MAX operator scans its state and expires all tuples with expiration timestamp T_1 and produces a new output after processing each expired tuple.

6.1.3 Reducing the number of join messages

One problem in the join message approach as described in the previous section is that if the join operator sends a join message for every incoming negative tuple, then unnecessary messages may be sent even if their corresponding positive tuples have not produced any join results before. This happens when the join filter is highly selective (i.e., when most of the input tuples do not produce join outputs). In this section, we propose techniques to send only the join messages necessary. Join operators can be classified into the following two classes: (1) joining a stream with a table, and (2) joining two streams.

Joining a stream with a table: In this case, only stream tuples will have negative counterparts. To process the negative tuples efficiently, the join operator will keep a table (Joined Tuples Table, JTT) in a sorted list (sorted on the timestamp). When a positive tuple is processed and produces join results, then the timestamp of this positive tuple is entered in JTT. At most, the size of this table is equal to the window size.

When a negative tuple is to be processed, the join checks whether there is a tuple, say t, with the same timestamp in JTT. If found, this means that t has produced join results previously, and a join message will be sent for t and t's timestamp is removed from JTT. If the tuple timestamp is not in JTT then the negative tuple is simply ignored. Notice that a join message is more beneficial in

Algorithm 5 The Modified W-Join Algorithm
Input: t_i : *Incoming tuple from stream* S_i. H_1, H_2: *Hash tables for* S_1 *and* S_2 *represent the join operator state.*
Algorithm

1) If t_i *is a positive tuple*
2) $B_x = hash(t_i)$
3) Insert t_i *in the bucket* B_x *in the hash table* H_i
4) For each tuple t_j *in bucket* B_x *in the other hash table*
5) If t_j *joins with* t_i
6) output a positive join output tuple t^+ *for* (t_i *and* t_j) *with:*
7) $t^+.ts = max(t_i.ts, t_j.ts)$
8) $t^+.Ets = min(t_i.Ets, t_j.Ets)$
9) If ($t_j.Ets < t_i.Ets$)
10) Increment reference count of t_j *by one*
11) Else Increment reference count of t_i *by one*
12) Else if t_i *is an expired tuple*
13) $B_x = hash(t_i)$
14) Delete the tuple t_i *from the bucket* B_x
15) If reference count of $t_i > 0$
16) Send a join message with timestamp $= t_i.Ets$

the case when a stream tuple joins with more than one tuple. The join message will be responsible for expiring all the join results produced previously.

Joining two streams: When the join operator joins two tuples t_i^+ from S_1 and t_j^+ from S_2, the resulting tuple t^+ should expire whenever either t_i^+ or t_j^+ expire. Assume that t_i^+ will expire first. To expire t^+, only the join message for t_i^+ is needed.

To avoid the unnecessary join messages, a reference count will be kept with every tuple t_x in the hash table. This reference count indicates the number of output tuples that should expire when t_x expires. The reference count of a tuple t_x is incremented by one when tuple t_x joins with tuple t_y and t_x has the minimum timestamp. The pseudo code for the join operator after adding the reference count is given in Algorithm 5. When the join operator is scheduled and a negative tuple is to be processed, the corresponding positive tuple is deleted from the hash table and the reference count associated with it is checked, if greater than zero then a join message for this tuple is emitted.

Example: Figure 8 gives an example on the reference count. When the join operator joins tuple t_i from Stream S_1(with timestamp T_1) with tuple t_j from Stream S_2 (with timestamp T_3), it will increment the reference count of t_i. At time T_6. tuple t_i from Stream S_1 expires. Since the reference count of t_i is one then a join message will be sent. No messages will be sent when t_j expires since its reference count is zero.

7 The Piggybacking Approach

As described in Section 4.2, the main motivation behind the NTA is to avoid the output delay that is incurred in the ITA. The output delay comes from either the low arrival rate or highly selective operators (e.g., join and select). Thus, in the case of high arrival rates and non-selective operators, the overhead of having negative tuples is unjustified. In fact, in these cases, the ITA is preferable over the NTA. In many cases, data stream sources may suffer from fluctuations in data arrival, especially in unpredictable, slow, or bursty network traffic (e.g., see [37]). In addition, due to the streaming nature of the input, data distribution is unpredictable. Hence, it is difficult to have a model for operator selectivity [27].

In this section, we present the *Piggybacking* approach for efficient pipelined execution of sliding-window queries. A similar notion of *piggybacking* is used in [1] to reduce storage needed to process a query. The main idea of the *Piggybacking* optimization is to self-tune the query pipeline by alternating between both the NTA and ITA. Thus, in the *Piggybacking* approach, negative tuples are processed only when they are needed. If positive tuples are flowing in the query pipeline with high rates (i.e., higher than the operators' processing rates), then every operator will always find positive tuples in its input queue. In this case, the positive tuples will carry the required information for expiration and there is no need for the negative tuples. In other words, there is no need to expire explicitly each tuple by having an explicit negative tuple. Instead, the information for expiring some tuples (timestamp) can be *piggybacked* with one of the incoming *positive* tuples. The *piggybacking* approach works in two stages:

Producing a *piggybacking* flag. When an operator produces an output tuple t (either *positive* or *negative*), it checks if there is any *negative* tuple in its output queue (i.e., the input queue of the next operator in the pipeline). If there is at least one *negative* tuple in the output queue, we perform two operations: (1) The output tuple t is tagged by a special flag $PGFlag$, (2) All the *negative* tuples in the output queue are purged. The timestamp of the tagged tuple is used in the second stage to direct the execution in the pipelined query operators.

Processing the *piggybacking* flag. When a query operator receives a tuple t (either *positive* or *negative*) at time T, it checks for the $PGFlag$ in t. If the input tuple is not tagged by the *piggybacking* flag, the query operator will act exactly as the NTA, described in Section 4.2.

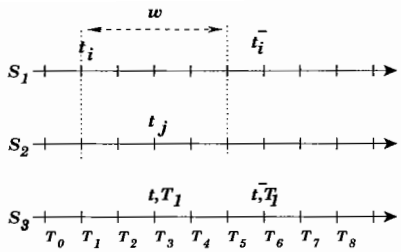


Figure 8: Reference Count Example.

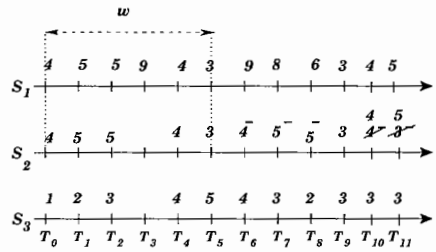
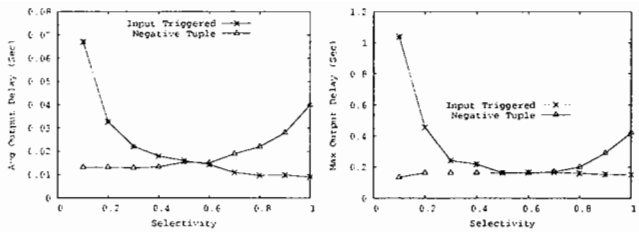


Figure 9: The Piggybacking Approach.

However, if the incoming tuple is tagged by the *piggybacking* flag, the query operator will act as the ITA, described in Section 4.1. This means that all tuples stored in the operator state with expiration timestamp less than T should expire (no negative tuples will arrive for these tuples later). Notice that waiting for a *negative* tuple for triggering expiration is performed only when there is a lack of inputs (either due to low arrival rates or due to selective operators). If there is plenty of inputs, then there is no need for processing *negative* tuples. In the case that processing the incoming tuple t does not result in any output (e.g., filtered with the select or join criteria), we output a null message that contains only the timestamp and the *piggybacking* flag so that operators higher in the pipeline behave accordingly.

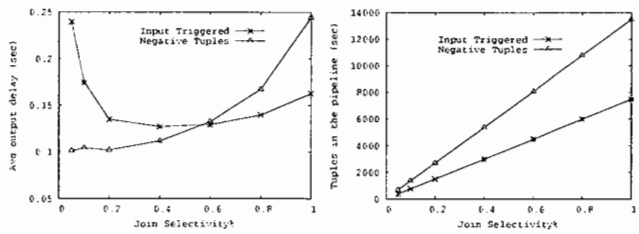
The *piggybacking* flag ($PGFlag$) is a generalization of the *join message* described in Section 6.1. The main difference is that a *join message* with timestamp T is responsible for expiring tuples with expiration timestamp T, while a $PGFlag$ with timestamp T is responsible for expiring all the tuples with expiration timestamps less than T.

Example: Figure 9 gives an example on the piggybacking approach. This example uses the same query of Figure 4a. The example shows that when the join operator is highly selective (in the period T_6 to T_8) negative tuples are passed to the COUNT operator for immediate expiration of tuples with values 4, 5 and 5. At time T_{10}, the join operator emits tuple 4^- immediately followed by tuple 4^+. If tuple 4^+ is emitted before the COUNT operator reads 4^-, then 4^+ will delete 4^- from the queue and COUNT will read only tuple 4^+. While processing 4^+, COUNT checks the input tuple's (4^+) timestamp and knows that a tuple with value 4 (that is stored in COUNT's state) should expire. Then, COUNT emits the new answer reflecting the expiration of 4 and the addition of 4. The same happens at time T_{11}. This example shows that the delay in the answer update will be the minimum possible delay.



(a) Avg Output Delay (b) Max Output Delay

Figure 10: Q1: Effect of Selectivity.



(a) Avg Output Delay (b) Tuples in pipeline

Figure 11: Q2: Effect of Selectivity.

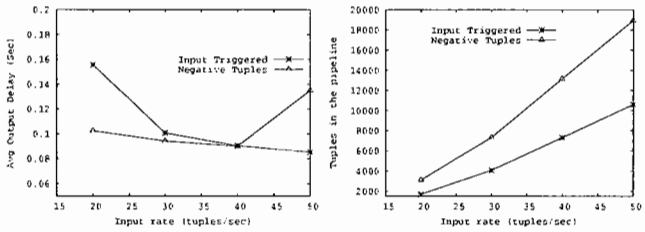
8 Experiments

In this section, we present experimental results from the implementation of our algorithms in a prototype data stream management system, NILE [24]. We compare the performance of the NTA with the ITA and show how the proposed optimizations enhance the performance further.

8.1 Experimental setup

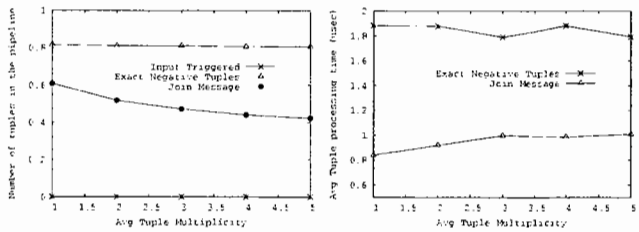
The prototype system is implemented on Intel Pentium 4 CPU 2.4 GHz with 512 MB RAM running Windows XP. The system uses the pipeline query execution model for processing queries over data streams. The query execution pipeline is connected with the underlying streaming source via the stream scan operator SSCAN. The `EXPIRE` operator is implemented as part of the SSCAN operator that is scheduled upon the arrival of input tuples. Different operators in the pipeline communicate with each other via a network of FIFO queues. Tuples are tagged with a special flag to indicate whether the tuple is positive or negative.

We use the average and max output delay as a measure of performance. The output delay is defined as the delay between the arrival/expiration of a tuple and the appearance of its effect in the query answer. For example, assume that in Q1 (Figure 4), a tuple $t1$ arrives to the system at time T. COUNT produces an output tuple after adding the value of $t1$ at time $T + d$, then this tuple encounters an output delay of d units of time. The stream *SalesStream* used in the queries has the same following schema: (StoreID, ItemID, Price, Quantity, Timestamp). We use randomly generated synthetic data. The timestamp follows the exponential distribution and is assigned to the tuple when the tuple arrives to the server.



(a) Avg Output Delay (b) Tuples in pipeline

Figure 12: Q2: Effect of Arrival Rate.



(a) Number of tuples (b) Avg processing time

Figure 13: Effect of the Join Message.

8.2 Input-triggered vs. negative tuples

In this section, we compare the performance between the input-triggered and the NTA for various queries. The comparison helps identify the appropriate situations to use each approach.

8.2.1 Effect of operator selectivity

Figure 10 gives the effect of changing the selectivity of the join operator in Q1 (Figure 4a). Figure 10a gives the average output delay while Figure 10b gives the maximum output delay. In this experiment, the input rate is fixed at 50 tuples/second, the window size is 30 minutes and the selectivity changes from 0.1 to 1. For low selectivity (less then 0.6), the ITA shows high output delay since the COUNT operator will not expire old tuples until a new input tuple passes the join. For the same range of selectivity (less than 0.6), the NTA does not depend on the selectivity since tuple expiration takes place even if no input tuples pass join. For higher selectivity (greater than 0.6), the ITA gives less output delay than the NTA. That is because, in the ITA, a large number of input tuples will pass the join and reach to the COUNT operator. The COUNT operator will be updating its state frequently with every input tuple. In the NTA, when the selectivity increases (greater than 0.6), then a large number of positive and negative tuples will pass the join and accumulate in the COUNT operator input queue. In this case, the output delay increases due to the waiting time in the queue.

Figure 11 gives the average output delay and the number of tuples in the pipeline join query Q2 (Figure 7a). The input rate is fixed to 50 tuples/sec for each stream and the join selectivity varies from 0 to 0.01. The window size is 30 minutes for each stream. The join selectivity is calculated

for each window. For example, the number of tuples in each window is 90K (50*60*30) tuples. The cartesian product of the two windows results in 8100 K tuples/window. When the join selectivity is 0.006, this means the number of tuples output of the join in each window is 48.6 K tuples (e.g., on the average the rate of the tuples output from the join will be 27 tuples/sec). Figure 11a shows that for periods of low join selectivity (less than 0.006), many input tuples will be filtered out by the join and will not reach the MAX operator. This causes the high output delay in the ITA. The NTA encounters less output delay since the negative tuples are sent out to update the query answer even if the input tuples are filtered out. As the join selectivity increases (greater than 0.006), the ITA gives less output delay than that of the NTA. That is because in the NTA a large number of positive and negative tuples will accumulate in the MAX operator input queue. The waiting time in the MAX input queue increases the output delay in the case of the NTA. Figure 11b shows that the NTA doubles the number of tuples in the query pipeline. The output delay can be acceptable as long as the MAX operator can still process the positive and negative tuples in a timely fashion. When the input rate of tuples to the MAX operator exceeds the operator's processing rate, tuples will accumulate in the input queue and cause the large output delay.

8.2.2 Effect of arrival rate

This experiment gives the effect of different input rates for query Q2 (Figure 7a). Figure 12 gives the average output delay and the number of tuples in the query pipeline for the two approaches. The join selectivity is fixed to 0.01 and the arrival rate varies from 10 to 50 tuples/second for each stream. The window size is set to 30 minutes. For low arrival rates (less than 40 tuples/sec), the ITA encounters higher output delays. The NTA encounters less output delays when the arrival rate increases. These results are the same as the results explained for Figure 11. We can see from Figure 11a that the ITA and the NTA have the same performance when the input rate is 50 tuples/second for each stream and the window is 30 minutes, and the selectivity is 0.006 (i.e., the output rate from the join is 27 tuples/sec). In Figure 12a, the two approaches have the same performance when the input rate is 40 tuples/second for each stream and the window is 30 minutes and the join selectivity is 0.01 (e.g., on the average the output rate of the join is 28 tuples/sec). The conclusion from this experiment is that as the rate of positive tuples flowing in the query pipeline increases, the ITA gives less output delay than the NTA.

8.3 The Join message optimization

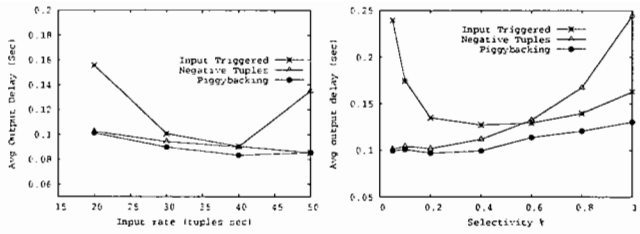
Figure 13 illustrates how the join message optimization reduces the overhead of processing negative tuples. This experiment uses Query Q2 (Figure 7a). The input rate is 50 tuples/second for each stream. The window is 30 minutes and the join selectivity is fixed to 0.01. The tuple's join multiplicity ranges from 1 to 5. To understand how to get different tuple multiplicity for the same join selectivity, assume the number of tuples in each window is 100, then for a join selectivity of 0.01, 100 tuples will be output from the join in each window (100/100*100). The 100 output tuples can result from 100 tuple from the first stream each joins from one tuple from the second stream (i.e., tuple multiplicity equals to 1). The 100 output tuples can also result from 50 tuples from the first stream each joins with 2 tuples from the second stream (i.e., tuple multiplicity equals to 2).

Figure 13a gives the ratio between the number of negative and positive tuples. The number of negative tuples represents the overhead associated with the NTA. This overhead is always zero for the ITA. The overhead is almost equal to one in the NTA since one negative tuple is processed for every positive tuple (in the figure, it is not exactly one since some negative tuples may have not been processed yet at the time measurement was taken). The join message optimization reduces the number of negative tuples emitted from the join operator to the next operator in the pipeline (MAX). The reduction increases as the tuple join multiplicity increases. Figure 13b gives the average processing time of a tuple in the query pipeline. The join message reduces the processing time to around half since the negative tuples do not perform the exact join.

Figure 13b illustrates that the average processing time per tuple is independent of the tuple multiplicity. In the symmetric hash join between two streams $S1$ and $S2$, an input tuple from stream $S1$ probes only one bucket in the hash table for stream $S2$ (see Section 5). The probing cost is negligible compared to the cost of doing the join and constructing the output tuple. In Figure 13b, the average processing time is independent of the tuple multiplicity because the join selectivity is fixed and the number of output tuples is independent of the tuple multiplicity.

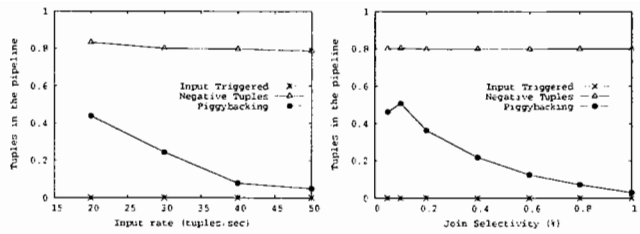
8.4 The Piggybacking approach

This section shows that the *piggybacking* optimization self-tunes the system to work in either the NTA or the ITA according to the system's load. In the following experiments, the piggybacking



(a) Effect of input rate (b) Join selectivity

Figure 14: Performance of Piggybacking.



(a) Effect of input rate (b) Join selectivity

Figure 15: Overhead of Piggybacking Approach

approach is accompanied with the join message optimization.

8.4.1 Performance enhancement

Figure 14 compares the output delay of the three approaches (ITA, NTA, and piggybacking) for various arrival rates and selectivities. Figure 14a is for Query Q2 (Figure 7a) when varying the input rates and fixing the join selectivity at 0.01 (the same input streams and settings as the experiment in Figure 12). Figure 14b is for the same query while varying the join selectivities with the input rate fixed at 50 tuples/second (the same input streams and settings as the experiment in Figure 11). The figure illustrates that for low arrival rates or high selectivity, the NTA gives less output delay since the expiration is performed independent from the arrival of new input tuples. On the other hand, when the input arrival rate increases, the NTA encounters more output delays since the queues are flooded by positive and negative tuples. The *piggybacking* approach gives the minimum possible output delay in all arrival rates and all selectivities since it processes the negative tuples only when necessary.

8.4.2 Reducing overhead

This experiment shows how the piggybacking approach reduces the number of negative tuples processed by operators in the pipeline. Figure 15 gives the ratio between the number of negative tuples and the number of positive tuples processed by the MAX operator in Query Q2. We vary the arrival rate and join selectivity in Figure 15a and Figure 15b, respectively. For the ITA, the ratio is always zero since there are no negative tuples processed. In the NTA, the ratio is almost

one since one negative tuple is processed for every positive tuple. In the piggybacking approach, the ratio decreases for higher input rates and higher selectivity. The reason is that positive tuples flow in the query pipeline with high rate and hence purge negative tuples (if any) from the queue before they get processed.

9 Conclusions

Negative tuples have been adopted by data stream management systems as a coordination scheme among various pipelined query operators. In this paper, we put the *negative tuples* under a magnifying glass where we show its detailed realization in terms of its generation and its processing in various operators. We presented algorithms for the different window operators. The difference between the window operators and the traditional operators is that the window operators have to processes expired tuples as well as newly incoming tuples. We show that although *negative tuples* avoid the shortcomings of the traditional *input-triggered* approach (e.g., large output delays), *negative tuples* suffer from a major drawback. *Negative tuples* double the number of tuples in the query pipeline, hence the pipeline bandwidth is reduced to half. Thus, we presented two optimization techniques that enhanced the performance of the *negative tuples* approach. The first optimization, namely the *join message* optimization, is concerned with the *join* operator subtree. The main idea is to filter out some of the *negative* tuples from the join operator input queue, thus, avoiding to re-execute the expensive join operation. The second optimization, namely the *piggybacking optimization*, self-tunes the query pipeline to work in either the *input-triggered* or the *negative tuples* schemes according to the rate of the tuples flowing in the query pipeline. With *piggybacking*, the query pipeline gets the benefits of both the traditional *input-triggered* and the *negative tuples* approaches. Both optimizations can be applied independently or together to enhance the performance of *negative tuples*. Experimental results based on a real implementation of *input-triggered*, *negative tuple*. *join messages*, and *piggybacking* inside a prototype data stream management system show that the join message optimization enhances the performance of *negative* tuples by a factor of two. Based on the input rate and/or join selectivity, the *piggybacking* optimization always traces the best performance of either the *negative tuples* or the *input-triggered* approaches.

10 Acknowledgment

This work was supported in part by the National Science Foundation under Grants IIS-0093116, IIS-0209120, and 0010044-CCR.

References

[1] D. Abadi. et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[2] D. J. Abadi. et al. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2):120–139, 2003.

[3] A. Arasu. et al. STREAM: The Stanford Data Stream Management System. *Book Edited By Garofalakis, Gehrke and Rastogi*, 2005.

[4] A. Arasu, S. Babu, and J. Widom. CQL: A Language for Continuous Queries over Streams and Relations. In *DBPL*, 2003.

[5] A. Arasu and J. Widom. Resource Sharing in Continuous Sliding-Window Aggregates. In *VLDB*, 2004.

[6] A. Ayad and J. F. Naughton. Static Optimization of Conjunctive Queries with Sliding Windows Over Infinite Streams. In *SIGMOD*, 2004.

[7] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *PODS*, 2002.

[8] S. Babu and P. Bizarro. Adaptive Query Processing in the Looking Glass. In *CIDR*, 2005.

[9] S. Babu, P. Bizarro, and D. DeWitt. Proactive Re-optimization. In *SIGMOD*, 2005.

[10] S. Babu, U. Srivastava, and J. Widom. Exploiting k-Constraints to Reduce Memory Overhead in Continuous Queries over Data Streams. In *TODS*, 2004.

[11] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *SIGMOD*, 1986.

[12] S. Chandrasekaran. et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[13] S. Chandrasekaran and M. J. Franklin. Streaming Queries over Streaming Data. In *VLDB*, 2002.

[14] S. Chandrasekaran and M. J. Franklin. PSoup: a system for streaming queries over streaming data. *VLDB Journal*, 12(2):140–156, 2003.

[15] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, pages 379–390. 2000.

[16] L. Ding, N. Mehta, E. A. Rundensteiner, and G. T. Heineman. Joining Punctuated Streams. In *EDBT*, pages 587–604, 2004.

[17] J. Gehrke, F. Korn, and D. Srivastava. On Computing Correlated aggregates over Continual Data Streams. In *SIGMOD*, 2001.

[18] L. Golab and M. T. Ozsu. Issues in Data Stream Management. *SIGMOD Record*, 32(2), June 2003.

[19] L. Golab and M. T. Ozsu. Processing Sliding Window multi-joins in Continuous queries over Data Streams. In *VLDB*, 2003.

[20] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[21] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *SIGMOD*. 1995.

[22] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[23] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.

[24] M. A. Hammad. et al. Nile: A Query Processing Engine for Data Streams (Demo). In *ICDE*, 2004.

[25] M. A. Hammad, W. G. Aref, and A. K. Elmagarmid. Stream Window Join: Tracking Moving Objects in Sensor-Network Databases. In *SSDBM*, 2003.

[26] M. A. Hammad, T. M.Ghanem, W. G. Aref, A. K. Elmagarmid, and M. F.Mokbel. Efficient Pipelined Execution of Sliding Window Queries over Data Streams. In *Purdue University Technical Report, CSD TR 03-035*, June 2004.

[27] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An Adaptive Query Execution System for Data Integration. In *SIGMOD*, pages 299–310, 1999.

[28] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating Window Joins over Unbounded Streams. In *ICDE*, 2003.

[29] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. Tucker. Semantics and Evaluation Techniques for Window Aggregates in Data Streams . In *SIGMOD*, 1995.

[30] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously Adaptive Continuous Queries over Streams. In *SIGMOD*, pages 49–60, 2002.

[31] R. Motwani. et al. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *CIDR*, 2003.

[32] E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. CAPE: Continuous Query Engine with Heterogeneous-Grained Adaptivity (Demo). In *VLDB*, 2004.

[33] U. Srivastava and J. Widom. Flexible Time Management in Data Stream Systems. In *PODS*, 2004.

[34] U. Srivastava and J. Widom. Memory-Limited Execution of Windowed Stream Joins. In *VLDB*, pages 324–335, 2004.

[35] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous Queries over Append-Only Databases. In *SIGMOD*, pages 321–330, 1992.

[36] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *TKDE*, 15(3):555–568, 2003.

[37] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost Based Query Scrambling for Initial Delays. In *SIGMOD*, pages 130–141, 1998.

[38] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. In *VLDB*, 2003.

[39] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic Plan Migration for Continuous Queries Over Data Streams. In *SIGMOD*, 2004.